LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-502372

On the role of NVRAM in data intensive HPC architectures

B. Van Essen, R. Pearce, S. Ames, M. Gokhale

September 30, 2011

IEEE International Parallel & Distributed Processing Symposium
Shanghai, China
May 21, 2012 through May 25, 2012

Disclaimer

On the role of NVRAM in data-intensive architectures: an evaluation

Brian Van Essen† Roger Pearce†‡ Sasha Ames† Maya Gokhale†
†*Center for Applied Scientific Computing*
Lawrence Livermore National Laboratory
Livermore, CA 94550
{*vanessen1, pearce7, ames4, gokhale2*}*@llnl.gov*

‡*Department of Computer Science and Engineering, Texas A&M University*

***Abstract*—Data-intensive applications are best suited to high-performance computing architectures that contain large quantities of main memory. Creating these systems with DRAM-based main memory remains costly and power-intensive. Due to improvements in density and cost, non-volatile random access memories (NVRAM) have emerged as compelling storage technologies to augment traditional DRAM.**

This work explores the potential of future NVRAM technologies to store program state at performance comparable to DRAM. We have developed the PerMA NVRAM simulator that allows us to explore applications with working sets ranging up to hundreds of gigabytes per node. The simulator is implemented as a Linux device driver that allows application execution at native speeds. Using the simulator we show the impact of future technology generations of I/O-bus-attached NVRAM on an unstructured-access, level-asynchronous, Breadth-First Search (BFS) graph traversal algorithm.

Our simulations show that within a couple of technology generations, a system architecture with local high performance NVRAM will be able to effectively augment DRAM to support highly concurrent data-intensive applications with large memory footprints. However, improvements will be needed in the I/O stack to deliver this performance to applications. The simulator shows that future technology generations of NVRAM in conjunction with an improved I/O runtime will enable parallel data-intensive applications to offload in-memory data structures to NVRAM with minimal performance loss.

***Keywords*-data-intensive; memory architecture; NVRAM;**

I. Introduction

Data-intensive applications with large working sets challenge the capacity of present day computing systems. To better accommodate data-intensive applications, recent supercomputer architecture proposals (*e.g.* [1]) advocate integrating non-volatile storage into the fabric of computing systems. In our work, we focus on the architecture of a single multi- (many-) core compute node with one or more PCIe-attached high performance NVRAM storage array(s). This configuration offers numerous benefits both standalone and even more as part of a compute cluster, including reduced operating costs for power and cooling, greater reliability, and potentially, higher performance.

It is often the case that the number of nodes in a supercomputing cluster is dictated more by the amount of memory than by compute power. If NVRAM can effectively increase the memory available to the application without appreciably increasing power requirements, clusters could achieve desired performance at lower node count, reducing cost and power/cooling requirements. Alternatively, individual NVRAM-augmented compute servers can scale to tackle extremely large data-intensive problems, as demonstrated by the 7th highest ranked machine in the Graph500 for June 2011 [2].

In this work, we address the challenge of achieving DRAM-like performance when some portion of the program state resides in I/O bus-connected storage arrays capable of low latency random access. We use the programming model proposed by other researchers of storage class persistent memory (*e.g.* [3] and [4] among others) in which all program state is addressed as in-memory, and some portion of that state resides in NVRAM. There are several variants of that model ranging from using DRAM as a cache for the large persistent memory, to providing language constructs that manually partition the program state between persistent and transient data.

As an implementation mechanism in Unix-like systems, the most natural method is to memory-map the contents of the NVRAM into the application's address space. Not only is the integration to the application simple, but the programmer interacts with the persistent data in the same manner as dynamically allocated stack and heap data structures. Furthermore, the operating system provides automatic runtime buffer management for the data stored in NVRAM. We choose a memory-mapped abstraction in preference to standard buffered I/O or direct unbuffered I/O for its potential as a high-performance interface and for the ease with which it could be integrated into legacy codes.

In this paper, we use parallel, asynchronous graph traversal as the driving application. We have found that large graph analysis problems are a very compelling example for data-intensive computing. Specifically, Breadth-First Search graph traversal combines several different and very challenging data access patterns that are quintessentially data-intensive computing. At the same time, it is a fundamental analysis capability that many researchers are interested in solving efficiently. Our previous work [5] has shown that

for this application, a multi-core compute node with NAND Flash memory can effectively replace a collection of traditional DRAM-only compute nodes.

Graph analysis presents extreme demands to the memory hierarchy. At their core, many graph analyses create unstructured memory access patterns, as exemplified by standard traversals such as Breadth-First Search (BFS). An unstructured access pattern defeats many of the traditional optimizations for rotating media and the current generation of NAND Flash [6]. We have found that algorithmic techniques such as high thread concurrency, with CPU oversubscription, can mitigate the access latency for current NVRAM. In this work, we focus on the per-node design pattern used in our graph traversal algorithms: highly multi-threaded, memory-mapped access to a node-local, I/O-bus-attached NVRAM subsystem.

Several recent research efforts have demonstrated the feasibility of new types of non-volatile random access memory (NVRAM) such as phase-change memory (PCM), spin-torque transfer ram (STT-RAM), and memristors. These technologies are predicted to provide lower overall energy consumption, demonstrate read and write latencies that approach DRAM, and support efficient random read/write access patterns. In this work we show how improvements in read and write latencies of future NVRAM technologies will impact algorithms with memory access patterns that are predominantly unstructured. Furthermore, we demonstrate that with proper latency hiding techniques, it is not necessary for NVRAM to achieve raw DRAM-class performance to provide compelling system performance. Finally, we demonstrate that the combination of high-performance NVRAM and an efficient memory-mapped runtime will allow applications to effectively offload heap allocated data structures into persistent storage, reducing the application's DRAM requirements.

We consider the impact of these new NVRAM technologies as fast peripheral-attached storage, as this is the most likely form factor to appear in the five year time frame. In this model, NVRAM appears as traditional persistent block storage. It is connected to the CPU via a peripheral bus and is part of a name/address space that is physically separate from main memory. To model these technologies and their impact on data-intensive applications at scale we have developed the PerMA simulator, which is detailed in Section III. In our use cases, the application accesses data associated with named files in a filesystem by memory-mapping the files using the Linux `mmap` system call interface. We favor this method of interaction because it provides naming, sharing, and protection through the file system, and simultaneously provides in-memory access through virtual address mapping. For this work we do not explore memory architectures that place NVRAM on the memory bus (*e.g.* [7]), as we see this architecture appearing at the cost and density we need beyond our relatively near term horizon [8]. Rather, our analysis studies future device technologies with lower latency than NAND Flash accessed over faster peripheral buses than currently exist.

II. Future NVRAM technologies

NAND Flash memory is currently the leading mass produced NVRAM technology. It is characterized by asymmetric read, write, and erase times and only provides access to page and block sized regions. In general, it requires 10s of microseconds to read, 100s of microseconds to write, and 1000s of microseconds to erase blocks of data. These devices typically perform hundreds to thousands of times faster than traditional spinning media, especially for random access workloads, but at the same time, lag DRAM latencies by several orders of magnitude.

The next generations of NVRAM technologies may still have asymmetric read / write / erase times, but are predicted to be faster than Flash by an order of magnitude or more. These technologies provide lower average power than DRAM by storing data as something other than the presence or absence of an electrical charge. For example, these new technologies store data as a crystalline state, magnetic spin, or oxygen vacancies, avoiding the cost of data refresh and static leakage. Furthermore, these storage techniques are predicted to scale to higher densities than traditional DRAM, offer read / write access times that are closer to DRAM speeds, and provide a finer granularity of access than NAND Flash. This combination of improvements makes these NVRAM technologies compelling for augmenting traditional DRAM when supporting data-intensive applications.

While many of these technologies exist in research labs, none of them are yet available at the scale or density required to support data-intensive computing. Therefore, to study the impact that these future technologies will provide for system architects and programmers, we have developed a simulator that allows us to model the read and write access times of these future technologies. Our simulator runs at a scale that allows us to explore interesting data-intensive applications.

III. The PerMA simulator

The Persistent Memory Architecture (PerMA) simulator is designed to model a system architecture that integrates high performance, page accessible, persistent memory into the address space of a process. The simulator supports hundreds of gigabytes of storage and can simulate the predicted delays for a wide range of NVRAM technologies and generations. We developed the PerMA simulator as a device driver for Linux. It is a ramdisk-style device that stores an application's data in privately held DRAM pages, and can insert a customizable delay when a page of data is accessed. This simulator is open source and is available at [9] for non-commercial public and academic use.

The PerMA driver can insert distinct access delays for both page read and write, and can model the overhead of

moving data through a peripheral bus and a device controller. The driver is re-entrant and scales to support hundreds of concurrent page requests. Furthermore, the driver can support application execution in real-time.

A. Simulation Model

One of the main design goals of this simulator is to provide a realistic, but not overly complicated performance and delay model. In particular, the model was designed to characterize the major performance factors in the operating system, peripheral bus, and NVRAM device, without trying to specifically replicate a single device or system. This approach allows us to provide a first-order approximation of an application's performance, and serve as a realistic upper bound on hardware performance, assuming that software bottlenecks can be addressed. The key characteristics of the performance model, shown in Equation 1, are listed below and are customized for read and write access.

- Let D be the device access latency, which is the time required to access a block of NVRAM technology, typically 4 KiB.
- Let P be the peripheral bus latency per byte, the time required to transfer a single byte of data across the peripheral bus. Note that if this value is 0, then the NVRAM is effectively attached via a memory bus.
- Let E be the efficiency of the peripheral bus, incorporating protocol, software driver, and hardware controller overheads.
- Let C be the device controller latency per byte, the time required by the NVRAM device controller to process each request and send or receive the data.
- Let R be the driver latency, *i.e* the amount of time required by a software driver to process the request prior to queuing a request to the peripheral bus.
- Let N be the number of concurrent read or write accesses.
- Let S be the size of the block of data being transferred, typically a single 4 KiB page.

$$IO\_latency = D + R + N * (P/E + C) * S \quad (1)$$

The performance model is designed to simulate a peripheral bus that allows pipelined access and a NVRAM architecture with abundant parallelism internally for servicing requests. The peripheral I/O bus model is a packet based protocol, with a fixed cost for sending an individual packet of data. Therefore the time required to service any individual read or write access is proportional to the number of concurrent requests already outstanding and the base access time that results from the device access latency and driver latency. The driver also internally models a page/buffer cache that can be limited to a programmable fixed size.

B. Implementing the PerMA simulator

The PerMA device driver is implemented as a character device for the Linux 2.6.x kernels, which allows it to be directly memory-mapped into an application's virtual memory space. Directly memory-mapping the device driver allows for a very lightweight and scalable interface for the application. Furthermore, once a page is memory-mapped into the virtual address space, it is accessed at hardware speeds. An alternative approach would be to use the block device interface. However, we found that a driver based on the Linux block ramdisk did not scale as effectively to highly concurrent access patterns as the memory-mapped character device.

The basic mechanism for instituting the $IO\_latency$ starts by memory-mapping a file or the raw device into the application's virtual memory address space. Then any access to data held by the PerMA driver will potentially trigger a page fault, that the PerMA driver will service. The driver will map the desired page into the user's page table, but it will also force the application thread to delay its execution to model the time required to fetch the page from hardware in the simulated system. This delay is implemented as a spin-wait, schedule yield, or a sleep depending on the desired length of the delay. For the majority of the simulated latencies we have found that forcing the thread to yield the processor without going to sleep provides the best balance of minimizing overhead and achieving an actual delay that is close to the desired delay.

Since the driver is re-entrant, when concurrent accesses are issued by independent CPU cores, they each calculate the expected delay (Eqn. 1) and then wait until that time has elapsed. Using this approach, independent requests are able to overlap their device access latency (D) and driver latency (R), which corresponds to having parallel access to the hardware and independent cores. The transit delay ($P/E + C$), which corresponds to the peripheral bus latency and device controller latency, is then proportional to the number of concurrent accesses to those shared resources.

One challenge when using a memory-mapped interface is that once a page mapping is established, the PerMA device driver normally does not see any subsequent requests for that page. Therefore, without further action, a memory-mapped device driver cannot model a fixed sized buffer. The simulator solves this problem by implementing a page management runtime that provides a simple page eviction policy and ensures that only a fixed number of pages are simultaneously mapped into the application's address space. The PerMA driver does this by using first-in-first-out (FIFO) queues that track when a page fault occurs and periodically flushing all page table mappings for small batches of the oldest pages (*i.e.* least recently faulted) in the queues. This forces subsequent accesses to these pages to incur page faults. Faults for pages that are in the process of being

flushed from the page tables (*i.e.* eviction), are minor faults and will recover the page from an eviction queue with minimal delay. Access to new pages, or ones that have completed being evicted from the queues, will incur a full page fault with associated *IO_latency* to model bringing the data from NVRAM.

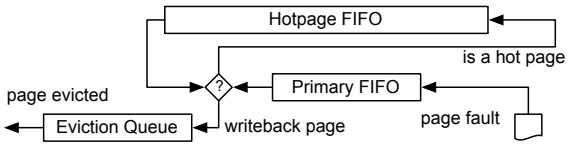

Figure 1. PerMA buffer management.

Figure 1 shows a logical diagram of the FIFO queues in the driver. When a page fault occurs, the simulator models the read time and then inserts the page into the primary FIFO. In the steady state, this insertion will force a page to be removed from the primary FIFO. If the page that is removed from the primary FIFO has experienced a greater than average number of page faults, it is hot and will be placed in the hotpage FIFO at which point a page from the hotpage FIFO is placed in the eviction queue. Otherwise, the page is placed in the eviction queue. As a page is queued up to be evicted it is removed from the application's page tables and the simulator models the writeback to persistent storage for all dirty pages. Finally, groups of pages in the eviction queue are then removed from processor's translation lookaside buffers (TLBs) in bulk.

IV. Related Work

There have been several other research efforts that have explored the impact of future NVRAM technologies and the impact of Flash on applications and operating systems.

From an memory architecture point of view, the work by Qureshi et al. [7] provides a straightforward design for integrating PCM into the memory hierarchy. They use PCM with a small DRAM cache to improve both performance and lifetime of PCM. This style of architecture is one of the options for the PerMA simulator when it runs without the peripheral bus and associated controller.

The Moneta project [6], [10] examines the performance of fast NVRAM attached via a peripheral bus. They have developed an FPGA-based simulation platform that lets them provide precise timing characteristics for up to 64 GiB of simulated memory. While this platform provides an effective mechanism for exploring future NVRAM technology, it is limited to today's bus technology and does not scale to data-intensive applications with workings sets in the hundreds of gigabytes.

The Mnemosyne [3] project considers how to effectively write to NVRAM devices. Both our and their work use a device simulator, but they base their simulator on the native block ramdisk present in the Linux 2.6 source code. Like the PerMA simulator, the device driver is able to delay read and write access to pages of memory, although the block driver is restricted to working with standard and direct I/O accesses. We previously explored the same line of implementation, but we found that the Linux block ramdisk driver is not as scalable to a large number of threads at high rates of access as a tuned memory-mapped driver. Furthermore, the driver is only able to catch I/O requests that are not serviced by the page cache.

SSDalloc [4], explores mechanisms for making allocation and write-back of persistent objects more efficient, but does not explore the performance impact of NVRAM. Their goal is to make SSDs appear more like memory, while PerMA preserves some aspects of a file system, so persistent objects can be named like files. On the other hand, BPFS [11] exposes PCM through a file system interface and considers optimizations for a hardware architecture to support efficient I/O, but they do not support memory-mapped access to files stored in PCM.

V. Graph Analysis - BFS Traversal

As mentioned previously, graph analysis is one of the driving applications for data-intensive computing. Traversing large graphs typically produces an unstructured sequence of memory references, with very little locality, and a low computing to communication ratio. The poor computation to communication ratio makes it difficult to efficiently parallelize a graph traversal across a HPC system, thus making data-intensive architectures with local NVRAM attractive since they have lots of low-latency storage that is shared between cores and processors.

The implementation of Breadth-First Search that we are using was derived from a previously published version in [5]. In our previous work, we demonstrated that our level-asynchronous multi-threaded approach to graph traversal is capable of tolerating I/O latencies when the graph is stored on NAND Flash. Additionally, we showed that with sufficient oversubscription of thread-level parallelism our semi-external graph algorithm, with the graph stored in NAND Flash, was within $2-3\times$ of the performance when the graph was stored completely in DRAM. To achieve this, we manually partitioned the associated Breadth-First Search data structures into heap allocated algorithmic data and persistent memory allocated graph data; the heap allocated algorithmic data represents the minority of the total data, but the majority of the total accesses and has access patterns that are the most unstructured.

In this work we extend that experiment to find out how much more performance improvement from the NVRAM is necessary to close that performance gap, and if that performance comes directly from the NVRAM access latency reduction, improvements in peripheral bus bandwidth, or other factors. Furthermore, we test the impact of reducing the size of the working set for the application by offloading

heap allocated data structures to persistent storage. Thus, in contrast to our previous work where data structures were manually partitioned into heap and persistent memory, in this work we allocate all data structures into the persistent memory and allow the simulator to apply its page caching policies throughout the entire memory space. A challenge that arises from offloading the BFS algorithmic data to the persistent memory is that it is accessed much more frequently than the bulk of the graph data. Furthermore, within the BFS algorithmic data, while some small parts of the data structures are accessed the most frequently, the majority of the algorithmic data accesses have very little locality.

The asynchronous BFS application interfaces with the NVRAM using a memory mapped interface (`mmap`). In Section VI-B we identify bottlenecks with Linux's memory-map implementation. However even with these bottlenecks our previous work has shown that using `mmap` is faster than direct I/O requests for this application, when the application lacks internal page caching. This is due to the application's ability to reorder accesses to increase page reuse, which benefits the memory-mapped approach.

VI. Calibrating the Simulator

To establish that the PerMA device driver provides a reasonable model of target hardware technologies, we have run several calibration micro-benchmarks on the PerMA simulator and both FusionIO and Virident PCIe-attached Flash cards. The micro-benchmarks performed random read and write I/O operations to the memory-mapped PerMA driver or directly to the PCIe Flash devices. We chose to use a distribution of random addresses that was uniform across the entire address range. The single uniform random I/O workload is very challenging because it has minimal page locality and is comparable to the data access patterns for the vertex and edge data in asynchronous BFS graph traversal.

The test platform for the calibration and all experiments was a 4 socket, 32 core AMD Opteron machine with 512 GiB of DRAM. For calibration we used either a single 200 GiB SLC NAND Flash Virident tachIOn Drive PCIe 1.1 x8 card, or four 80 GiB SLC NAND Flash FusionIO ioDrive PCIe 1.1 x4 cards. The four FusionIO cards are configured as a single RAID-0 software raid, interleaved every 256 KiB. The BIOS on the system was configured to interleave the DRAM across all sockets and memory controllers. We disabled swap, as well as CPU throttling.

For the PCIe-attached Flash devices, we performed 4 KiB, page-aligned, direct I/O accesses. Direct I/O avoids the use of the buffer cache and associated software overheads, thus it provides a good estimate of the raw, achievable performance of the devices. For the PerMA driver, the simulated storage was interleaved across all NUMA nodes and the internal buffer was set at 2 GiB. This was large enough to provide a balanced sampling of the buffer management overhead and yet small enough to minimize the potential for caching effects. Furthermore, the results were manually inspected to ensure that fewer than $\sim 1\%$ of I/O requests hit in the cache. The results of the calibration are shown in Figures 2, 3, and 4, each of which plots the number of threads running versus the average number of I/O operations per second (IOPS) achieved. The micro-benchmarks, using a uniform random distribution of address requests, issued approximately 5,000,000 requests, which corresponds to approximately 20 GiB of data out of a 186+ GiB region in the persistent store.

A. Uniform random I/O distribution

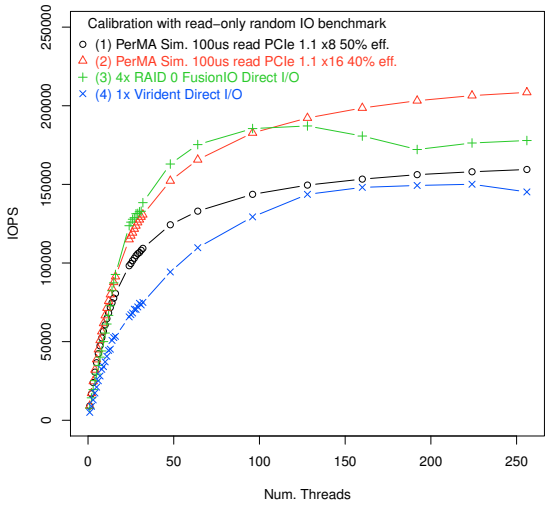

Figure 2. Read-only random I/O benchmark with uniform distribution. We refer to the curves shown here by the order in which they appear in the legend.

Figures 2, 3, and 4 show throughput measured in IOPS versus an increasing numbers of threads for read-only, write-only, and read/write request sequences, respectively. In Figures 2, 3, and 4 there are four configurations shown, two that used the PerMA driver, one with a single Virident card, and one with four RAIDed FusionIO cards. Curve 1 (the top line in the legend) shows the driver configured with a 8x PCIe bandwidth running at 50% efficiency, while curve 2 shows performance of the simulator with a PCIe x16 bus running at 40% efficiency. Finally, curves 3 and 4 show the performance of 4x FusionIO cards and a single Virident card, respectively. The efficiency rating of the simulator was derived empirically to match the first order performance of our target PCIe-attached Flash cards. The key differences between the empirical and theoretical PCIe bandwidth stem from the packet overhead on the bus and the PCIe software stack. The PerMA driver introduced no additional driver

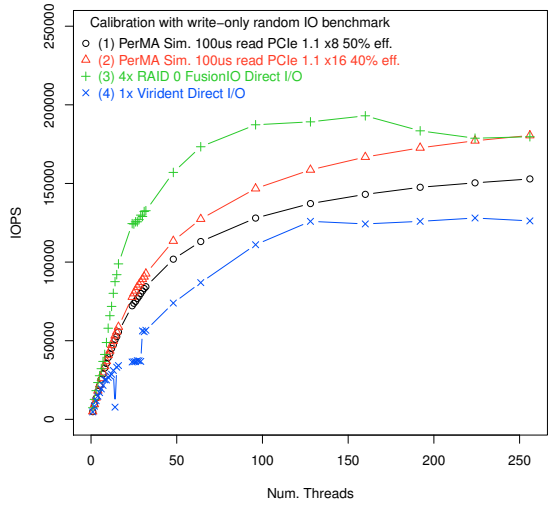

Figure 3. Write-only random I/O benchmark with uniform distribution. We refer to the curves shown here by the order in which they appear in the legend.

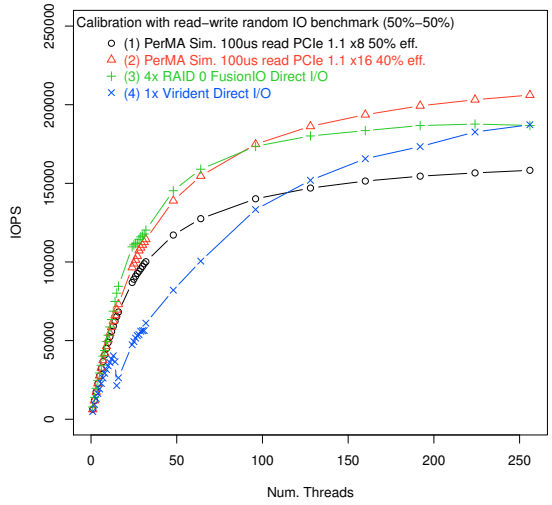

Figure 4. Read/write random I/O benchmark with uniform distribution. We refer to the curves shown here by the order in which they appear in the legend.

delay, beyond what was already required to service the page fault.

In Figure 2, curve 3 shows that the FusionIO read performance closely matches curve 2. Despite the theoretical bandwidth of the four RAIDed FusionIO cards, its performance is most closely matched by the simulator when modeling the 16 lane PCIe with 40% efficiency. The discrepancy between the stated and observed PCIe bandwidths is a product of how efficiently FusionIO's driver and controller is able to utilize the PCIe bus. Furthermore, we observe that after concurrency increases past 128 threads, FusionIO's throughput declines in contrast to the other curves. The PerMA driver specifically does not strive to exactly match the performance of the Flash cards because their results are a product of the complex interplay of OS I/O interface and scheduling, the device drivers, and the hardware controllers. Curve 4 (in Figure 2) shows the read performance of the Virident card, which does not exhibit the same drop in IOPS with increased concurrency. As a single device, it does not offer the same bandwidth as the RAIDed cards, however, it achieves a higher bus efficiency. As shown in Figure 2, its peak bandwidth comes close to that of curve 1 when the PCIe bus is modeled to achieve 50% efficiency.

Figure 3 shows the performance of the simulator and Flash cards for a write-only request sequence. The performance of both the FusionIO and Virident cards are much less predictable with write requests. Curve 3 shows that the IOPS achieved exceed the number predicted, peaking when there are tens to just over a hundred threads executing. Common optimizations such as early write completion (*i.e.* the write is cached and "completed" before it is fully flushed to the Flash cells) and I/O-bus request aggregation in the device driver or flash controller are the likely causes of this performance improvement. The performance of the Virident card, curve 4, has a very unusual, but repeatable dip with 14 threads and flatline performance with tens of threads. As with the read-only performance these types of irregularities are likely due to software and/or system idiosyncrasies and are not intended to be captured by the simulator, which is intended to give a first-order performance model.

Finally, Figure 4 shows the performance of the simulator and PCIe-attached Flash cards with a mixed 50%-50% read and write workload. Curve 3 shows that the performance of the four FusionIO cards is well modeled by the simulator as demonstrated by curve 2. The performance of the Virident card with a combined read and write workload (curve 4) is higher than its performance with either a read-only or write-only workload, which is attributable to the application of common optimization techniques in the Virident driver and storage controller.

Our experience with calibration shows that it is difficult to pin-point how a card is going to perform based solely on its specifications. FusionIO advertises a 26us access time for a 512B block. Depending on how the Flash is laid out, accessing a 4KiB block can take either 208us if each page is accessed serially, or 26us if they are all accessed in parallel. Furthermore, we have found that high-speed I/O is sensitive to variations between the CPU manufactures, CPU generations, motherboard topology, and versions of Linux. Our goal with this calibration was to tune the simulator to match the real-world performance that was observed in our laboratory's systems. However, a strength of our simulator

is that it can be tuned differently to predict the expected performance on other systems.

Based on our experience with these cards, estimating that they take around 100us to service a 4KiB page read or write is a reasonable approximation. Similarly, both types of PCIe-attached Flash cards only get roughly one half of peak bandwidth with either a single or multiple cards RAIDed together. As shown by the Moneta [6] research, this type of under-performance on PCIe is to be expected until the Linux I/O stack is overhauled to optimize data transfer for random access, fast, media. Going forward for all subsequent tests we use a modeled 50% PCIe bus efficiency.

B. *Identifying bottlenecks*

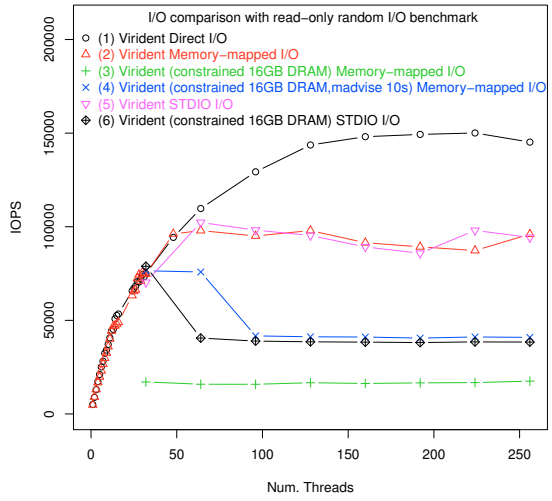

Figure 5. Comparing memory mapped access versus direct I/O.

The PerMA driver, which provides a memory-mapped interface, was calibrated by using direct I/O to measure the performance of the FusionIO and Virident PCIe-attached Flash cards, rather than memory-mapped I/O. The reason for the difference in approaches was that direct I/O captures the raw hardware performance of the PCIe-attached Flash cards better than OS memory-mapping (or standard buffered I/O) by avoiding the cache effects of the Linux page cache. Note that while we were able to successfully limit the size of the PerMA simulator's cache and thus avoid any caching effects, limiting the size of Linux's page cache introduces some performance problems that are illustrated here. These problems are a result of pushing the general-purpose software optimizations used in Linux in such a way that they become detrimental to the performance of data-intensive applications. Figure 5 shows the results of the uniform, random I/O benchmark when using direct I/O, standard I/O, and memory-mapped I/O to access a single Virident PCIe Flash card. The tests were conducted on a system with unconstrained memory (512 GiB DRAM), using read-only requests and a sampling of 20 GiB from a space of 186 GiB. Unlike the direct I/O (top curve), the second curve shows that the performance of memory-mapped I/O flattens out as the number of concurrent accesses increases. Similarly, the fifth curve shows that standard I/O also flattens out as concurrency increases. Clearly, this lack of scalability is a product of the software stack throttling performance. Pinpointing these choke points is the subject of future work.

Another challenge arises with both the memory-mapped and standard I/O in a constrained memory system, where the page (or buffer) cache cannot store the entire file and must evict pages. Curve 3 shows the performance of accessing 100 GiB of memory-mapped data on a system with only 16 GiB of DRAM. In this scenario, the page cache is forced to evict pages, and with a random memory access pattern, there are no pages that are clearly better or worse to remove. Figure 5 shows that the performance plummets when the system is under memory pressure. Note that the PerMA driver does not suffer from these limitations, since it is able to handle its own memory internally. For workloads with uniform, random access patterns and a working set that is much larger than the page cache, a partial solution for the constrained memory environment is to periodically use the Linux system call `madvise` with the `MADV_DONTNEED` flag. This system call tells the page cache that any page can be discarded, and as seen by curve 4 the performance improves when advised every 10 seconds. The performance of standard I/O (shown by curve 6) also suffers when the page cache is forced to evict pages in the constrained memory system.

In summary, we found that random access to Flash memory that was mapped in via the `mmap` system call incurred sufficiently high OS overhead that we couldn't measure the raw performance of the device. We therefore used direct I/O to avoid OS overhead while measuring latency and bandwidth on Flash. As shown in Section VIII, we also developed algorithms in the simulator tuned to the access patterns of our application as an initial experiment into efficient management of memory-mapped pages.

VII. Testing the Impact of Future NVRAM Technology

The goal of this work is to predict how significant of an impact different technological improvements will be on data-intensive applications with node-local NVRAM. As peripheral bus bandwidth increases and NVRAM access latency decreases, is application concurrency still important to system-level performance? Can an application using I/O-attached NVRAM achieve a level of performance that is competitive with a DRAM-based, in-memory algorithm? How fast do the NVRAM and I/O bus have to be to achieve such DRAM-class performance? Will the combination of fast NVRAM and an efficient memory-map runtime enable

Label	D (μs)	PCIe Gen.	# Lanes	E	P (ps/B)	C (ps/B)
L_{curr}	100	2.0	x8	50%	500	420
L_{nxt1}	100	2.0	x16	50%	250	420
L_{nxt2}	50	2.0	x32	50%	125	210
L_{nxt3}	25	2.0	x64	50%	62	105
Sim-opt	0	N/A	∞	N/A	0	0

Table I
READ CHARACTERISTICS FOR MODELED NVRAM TECHNOLOGY

systems to offload heap allocated structures into persistent memory, thus reducing the amount of system DRAM needed with a minimal loss in performance?

To address these questions we test our asynchronous BFS traversal algorithm with different levels of predicted technology improvements, shown in Table I. Through calibration, we arrived at an effective read and write latency of 100us for the current generation PCIe-based Flash cards. For future NVRAM access times we model a system with one-half (50us) and one-quarter (25us) the current access latency. Current generation bus technologies (*i.e.* PCIe 2.0) have already surpassed the performance of what we observed on the PCIe 1.1 cards that we had available. A realistic current generation bus would be PCIe 2.0 x8, and we use three future bus technologies with PCIe 2.0 x16, PCIe 2.0 x32 and PCIe 2.0 x64 for these projections. For the current generation's device controller latency (C) we used the number presented for the Moneta controller [6] that was built in FPGAs on the BEE3 platform, and halved it for each generation. As before the additional driver latency R was zero for these configurations. Finally, the sim-opt configuration will be used in later tests to determine the runtime system's overhead.

Figure 6 shows the performance of the random I/O read-write micro-benchmark on the current and three future generations of modeled NVRAM technology, providing an upper bound for how much improvement can be had from the technology scaling. We used 50 million I/O requests from the uniform random distribution since it is the most strenuous performance test, with the least opportunity for caching, and use a 90%/10% read/write mixture. The simulator was allocated a 2GiB primary FIFO and no hotpage FIFO to minimize cache effects.

The test was conducted with 32 threads (*i.e.* one thread per core) and 256 threads, which oversubscribed the system by 8×. When using 32 threads, the L_{nxt3} system serviced 4.20× the IOPS of the baseline system L_{curr}, and 3.49× of the baseline IOPS with 256 threads. With one half of the latency and two times the bandwidth, the largest gain in IOPS between tested generations was between L_{nxt2} and L_{nxt1}, which showed an improvement in IOPS of 1.95× and 1.92× for 32 and 256 threads, respectively. Overall, Figure 6 shows that for uniform random I/O, oversubscription still provides significant benefit as technology scales to the L_{nxt3} level, although at a slightly diminished level.

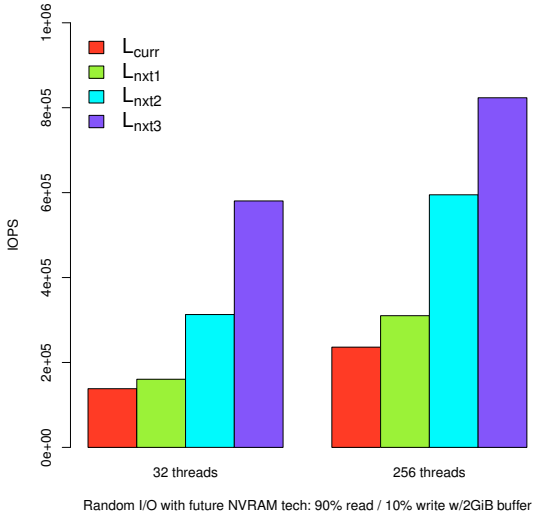

Figure 6. Measuring the potential impact of future NVRAM with random I/O benchmark.

VIII. MODELING LOCALITY OF ACCESS: CACHING A MORE COMPLEX I/O PATTERN

To the first-order, applications like BFS traversal that are dominated by unstructured memory requests are well characterized by a sequence of random I/O with uniform distribution. However, as we scale up the working set of this data-intensive application, and push against resource limitations in the real system or in the simulator, a more refined I/O model is desired. As described in Section V the data structures for the asynchronous BFS traversal are comprised of the algorithmic and the graph data. While each of these data structures is accessed in an unstructured pattern, the algorithmic data is accessed far more frequently. To characterize this more complex data pattern we use a partitioned, uniform random I/O distribution, where the address space is partitioned into two ranges: a small, frequently accessed range (*i.e.* hot pages) and a large, infrequently accessed range. Both subranges are sampled uniformly. Figure 7 shows the distribution of memory requests to the address space that occur in the asynchronous BFS traversal.

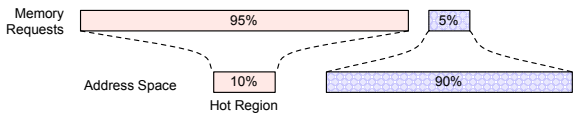

Figure 7. Partitioned, uniform, distribution of memory requests to address space.

A. Testing partitioned, uniform random I/O distributions

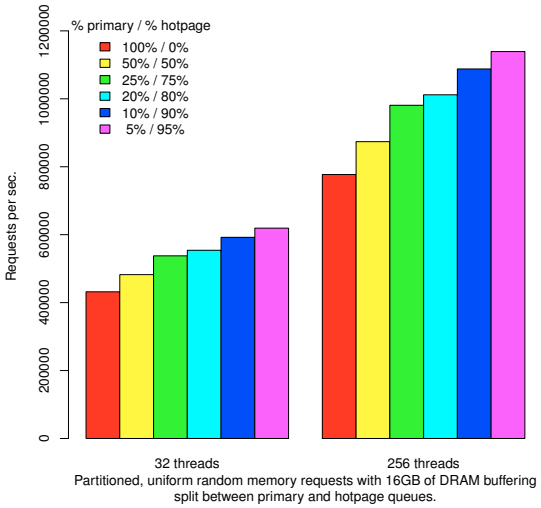

Figure 8. Read/write random I/O benchmark with partitioned, uniform distribution with 90% reads on the L_{curr} NVRAM generation. Note that the y-axis is in terms of memory requests issued.

The calibration of the PerMA simulator focused on tuning latency and bandwidth parameters of the simulator to real world PCIe-attached NVRAM using random I/O sequences with uniform distribution across the target address space. However, for asynchronous BFS traversal the distribution of the majority of memory requests to a small region of the address space is ripe for traditional techniques of caching and buffering. A rich exploration of the best caching techniques for this application is beyond the scope of this study. The experiments in this section focus on establishing a guideline for a simple, first pass optimization to provide some level of caching, and evaluating the optimization within the PerMA simulator. The memory request pattern used by this micro-benchmark is a partitioned, uniform distribution. As shown in Figure 7, this memory request distribution directs 95% of the memory request to only 10% of the address space. These memory requests are uniformly, but independently distributed for both the 10% of the address space that is hot, and the other 90% of the address space. This second micro-benchmark represents a memory request pattern that is demonstrated by our asynchronous BFS graph traversal and is more amenable to caching techniques than the uniformly random I/O sequence.

Page-level caching is implemented in the PerMA simulator by using a hierarchy of FIFO buffers (as shown in Figure 1), such that when a hot page is removed from the primary FIFO it is placed in the hotpage FIFO. This allows frequently faulted pages to remain within the buffer much longer than less frequently faulted pages. Using this simple hierarchy of buffers allows the simulator to easily trade-off how a fixed amount of system memory is utilized, shifting it between caching for hot pages versus the increasing the duration of buffering for new (or infrequently faulted) pages. The second micro-benchmark used the random I/O request generator with a partitioned, uniform random distribution where 95% of 150,000,000 memory request were uniformly distributed across 10% of a 177.6 GiB address space, while the remaining 5% of requests were distributed across the remaining 90% of the address space (Figure 7). Thus the micro-benchmark accesses approximately 46 GiB of data. The I/O requests were composed of a mixture of 90% reads and 10% writes to model the request pattern of our asynchronous BFS traversal.

The results of the second micro-benchmark are shown in Figure 8, which plots the number of threads running versus the average number of memory requests per second achieved. Figure 8 shows the performance improvement as a greater percentage of memory is allocated to the hotpage FIFO in a system where the PerMA simulator uses 16 GiB of DRAM for buffering and L_{curr} NVRAM technology. For simulations with both 32 and 256 threads, the number of requests serviced per second increases as more of the 16 GiB of DRAM is allocated to the hotpage FIFO, rather than the primary FIFO. Figure 8 shows that a partitioned, uniform random access pattern requires only a small amount of primary buffering to filter out the relatively infrequently faulted pages from the hot pages. This allows nearly all of the system's DRAM to be allocated for the hotpage FIFO. The opposing constraint for real world applications where there is some short term page reuse and the hotpage FIFO is smaller than the working set, is that a very small primary FIFO may allow hot pages to flush a normal page from the FIFO before it is finished being used. Furthermore, the frequent eviction of "normal" pages can cause them to become hot enough to put additional pressure on the hotpage FIFO. Therefore, we make the conservative decision for future tests to use 25% of DRAM for the primary FIFO and 75% of the DRAM for the hotpage FIFO.

IX. Results: BFS on future NVRAM generations

We performed experiments using our asynchronous BFS traversal on four simulated NVRAM devices, two Virident NAND Flash cards in a RAID 0 configuration, and DRAM only. Figure 9 shows the performance of our asynchronous BFS traversal using these configurations. The performance measure used is the Graph500 standard of *traversed edges per second (TEPS)*, where higher is better. To provide a realistic and challenging experiment, our experiments used the R-MAT [12] graph generator outlined in the Graph500 [13] benchmark. Using the generator, we created a scale-free graph with 2^{31} vertexes with an average out-degree of 16. The graph instance is labeled RMAT 31, and the data set for this graph is 146 GiB of vertex and edge data plus 24

GiB of BFS algorithmic data. The execution on the Virident PCIe-attached Flash cards used a total of 40 GiB of DRAM. The application allocated 24 GiB of BFS algorithmic data on the heap leaving 16 GiB for page cache. The experiments with the PerMA simulator were also given 40 GiB of DRAM but the BFS algorithmic data was memory mapped into the persistent region and the DRAM was split into 10 GiB for the primary FIFO and 30 GiB for the hotpage FIFO.

Figure 9 shows the performance of the asynchronous BFS traversal on real PCIe-attached NVRAM, in-memory, the raw simulator, the simulated devices L_{curr}, L_{nxt1}, L_{nxt2}, and L_{nxt3}. Examining the performance of the simulated devices we observe that concurrency is still critical for the projected technology generations, as the oversubscribed examples significantly out-perform those with only 32 threads. However, the trends in relative performance gain between technology generations is smaller with oversubscription than without. Specifically, the relative gain from L_{nxt2} to L_{nxt3} and L_{nxt3} to sim-opt is much smaller with 256 threads than with 32 threads. Furthermore, the TEPS that sim-opt achieved is not significantly higher with oversubscription than with 32 threads. These trends indicate that as peripherally-attached NVRAM becomes faster and "closer" to main memory, the benefit from oversubscription diminishes. Overall, these results highlight that a highly concurrent algorithm with asynchronous I/O can hide access latency though oversubscription, and that it will allow the algorithm to approach peak performance with less aggressive technology.

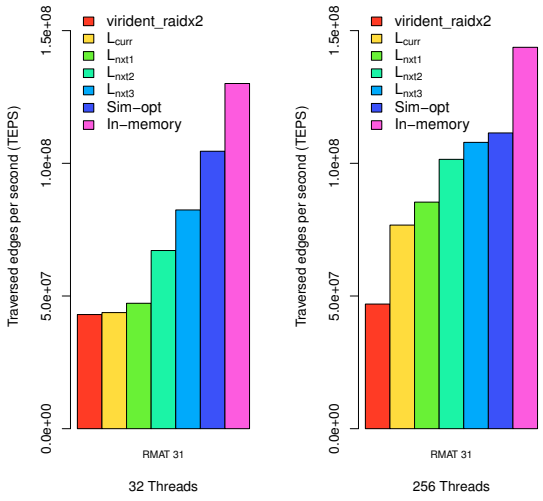

Figure 9. Measuring the potential impact of future NVRAM on the asynchronous BFS traversal. Showing graph size versus traversed edges per second.

The simulated device L_{curr} is configured to compare with the two Virident cards in a RAID 0 configuration. The similarity in performance between L_{curr} and the PCIe-attached Flash, when using 32 threads, demonstrates the fidelity of the simulator and its ability to model real world systems. However, the performance with PCIe-attached Flash and 256 threads was 0.61× slower than what was predicted by the simulator. The significant performance difference between L_{curr} and Virident at 256 threads shows the system software overhead (described in Section VI-B) when the application uses thread oversubscription and memory-maps the Virident devices. This application would see significant benefits through operating system improvements in the memory-map implementation when oversubscribing.

The right-most two bars of Figure 9 are the results for the simulator running with no delays as shown by the sim-opt bar, and the asynchronous BFS search running in-memory (all data stored on a tmpfs ramdisk). The in-memory (DRAM) performance is an upper bound for the system configuration, where further improvements cannot be achieved through improved NVRAM for this application. The sim-opt result shows the best possible result for running in the simulator, and approximates the upper-limit for any memory-mapped runtime since the majority of the buffer management tasks required by the simulator are also required for a memory-mapped runtime. The overhead of the simulator is 24% and 29% for 32 and 256 threads, respectively, as shown by the performance delta between the in-memory speed and sim-opt speed. Figure 9 shows that L_{nxt3} is able to effectively amortize nearly all NVRAM latency with 256 threads as it achieves 0.97× the TEPS of the sim-opt system, which has infinite bandwidth and no delays. Furthermore, it approaches DRAM speeds using 256 threads: 0.75× the TEPS of the in-memory system.

X. Results: BFS in constrained memory

As the problem size of data-intensive applications continues to grow, system memory (DRAM) is frequently a limiting factor in the size of problem that we are able to tackle. By moving data structures in the asynchronous BFS traversal that are traditionally heap allocated into the persistent memory we are able to test the resilience of this latency tolerant algorithm to constraints on the amount of main memory available. Figure 10 show the effects of reducing the available system DRAM when using one thread per core and thread oversubscription. Performance is measured in TEPS, and each curve is the performance of the asynchronous BFS traversal on a simulated technology generation. As noted earlier, the memory constraints from the previous test were 16 GiB page buffers and 24 GiB of BFS algorithmic data. Therefore, a system with 40 GiB of DRAM is the baseline, and the algorithm is tested with up to 42 GiB and down to 12 GiB, with 25% of the memory used for the primary FIFO and 75% for the hotpage FIFO. Figure 10 shows that as the system memory is constrained performance slowly drops off, where a system with 16 GiB of DRAM and L_{nxt3} NVRAM has 0.82× the TEPS of the

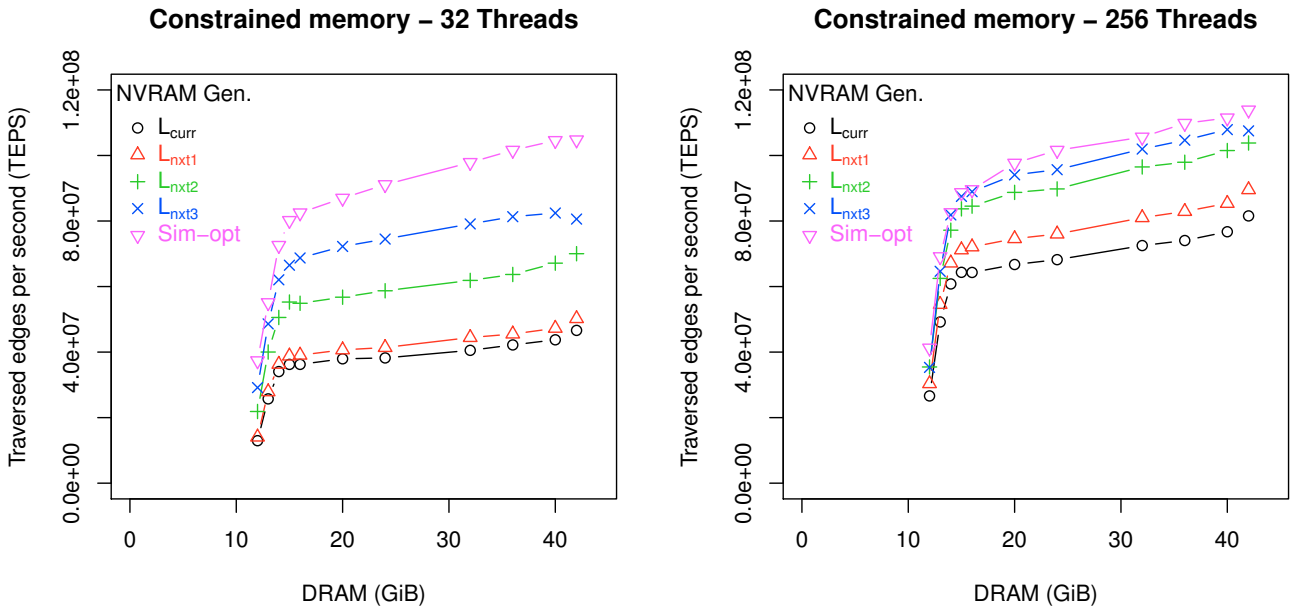

Figure 10. Measuring the potential impact of future NVRAM on the asynchronous BFS as system memory is constrained. Showing system DRAM versus traversed edges per second.

system with 40 GiB of DRAM. However, below 14 GiB of DRAM the performance declines rapidly. Additionally, adding more DRAM, for a system with 42 GiB, provides only a small performance benefit.

As before, the sim-opt curve has no delays or bandwidth limitations and thus represents the upper bound of the simulator's performance. Figure 10 shows that L_{nxt3} is within 3% of the simulator's raw performance with thread oversubscription, but is 27% slower than sim-opt when using only one thread per core. Overall, the combination of a latency tolerant algorithm and only modestly improved NVRAM performance can run effectively on a system with only 40% of the DRAM and suffer only a 21% reduction in performance. Figure 10 also shows a sharp drop in performance as the system's DRAM is reduced to 12 GiB. It is interesting to note that performance of each simulation drops dramatically, which indicates that this degradation is independent of NVRAM technology. Identifying the source of this performance cliff will be the subject of future investigation.

XI. Conclusions

The demands of data-intensive applications run counter to contemporary design trends for HPC architectures, where core count is dramatically outstripping main memory capacity and bandwidth. One hope for mitigating this trend is improvements in non-volatile random access memory technologies, specifically improved performance and density, and reduced energy consumption, making it feasible for NVRAM to augment DRAM in the memory hierarchy. We have developed the PerMA simulator that allows us to model the impact of future generations of I/O-bus-attached NVRAM on application performance at scale. The simulator provides a memory API to the persistent memory, models latencies and bandwidths ranging from current Flash down to DRAM-like performance, and supports hundreds of threads at native speed. We have calibrated the PerMA simulator to provide a first-order approximation of the performance of current generation PCIe-attached Flash cards. We then scaled down the NVRAM access times and peripheral bus latencies to model three combinations of future generations of technology improvements.

The simulator has provided new insights into the interaction of algorithmic techniques (*e.g.* thread oversubscription or out-of-core data structures) with future NVRAM technology generations as illustrated in an important benchmark application. Using the simulator, we have quantified the potential performance of an unstructured asynchronous BFS graph traversal algorithm. We have shown that with sufficient concurrency, this algorithm, using NVRAM, will be able to approach the performance of a fully in-memory algorithm. Using 40GB of DRAM plus Flash, the BFS algorithm could achieve 75% of the fully in-memory (170GB) performance on the most aggressive technology generation modeled.

Furthermore, this work illustrates that it may be possible to get scalable performance for out-of-core algorithms, with appropriate caching algorithms. Guided by the access patterns of asynchronous BFS, we have implemented a simple, two level buffering scheme within the simulator to provide

locality-optimized page mapping. We have shown that the combination of a latency tolerant, concurrent algorithm, future NVRAM devices, and an optimized memory-map runtime system enables migration of data structures that were traditionally heap allocated into persistent memory. This new environment will allow data-intensive applications both to scale to larger problem sizes without increasing main memory, or alternatively will allow for the same size problems to run in greatly constrained memory.

XII. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-502372). Funding partially provided by LDRD 07-ERD-063 and LDRD 11-ERD-008. Portions of experiments were performed at the Livermore Computing facility resources, with special thanks to Dave Fox and Ramon Newton. Pearce is supported in part by a Lawrence Scholar Fellowship and a Dept. of Education Graduate Fellowship (GAANN). At the time of this work Ames was a graduate student in the Department of Computer Science, University of California, Santa Cruz and supported by a Lawrence Scholar Fellowship.

References

[1] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 217–228.

[2] LLNL, "Kraken (1 node / 32 cores / FusionIO) Ranked 7th for Scale 34 @ 55.9 MTEPS," www.graph500.org, June 2011.

[3] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 91–104.

[4] A. Badam and V. Pai, "Ssdalloc: Hybrid ssd/ram memory management made easy," in *In Proc. 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[5] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 385–395.

[7] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, Jun 2009, pp. 24–33.

[8] A. Fazio, "Memory technologies," personal communication, Intel Corporation, 2010.

[9] "Data-centric Computing Architectures Research Group," https://computation.llnl.gov/casc/dcca-pub/dcca/Data-centric_architecture.html.

[10] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, Nov 2010, pp. 1–11.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Fourth SIAM International Conference on Data Mining*, April 2004.

[13] "Graph500," www.graph500.org.